Palo Alto Research Center

Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles

by Edward M. McCreight

XEROX

Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles

by Edward M. McCreight

CSL-80-9 JUNE 1980

© Xerox Corporation 1980

Abstract: See next page.

A version of this paper has been submitted to *SIAM Journal on Computing*.

CR Categories: 5.25, 3.74, 5.39

Key words and phrases: computational geometry, search trees, intersection, intervals, rectangles, concrete complexity.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

Abstract:

Let D be a dynamic set of intervals over the set 1,2,...,k of integers. Consider the following operations applied to D:

(1) Insert an interval [x,y] into D.

(2) Delete an interval [x,y] from D.

(3) Given a test integer u, list those intervals [x,y] in D that contain u.

(4) Given a test interval [u,v], list those intervals [x,y] in D for which [x,y] $\cap$ [u,v] is non-empty.

Using a new data structure that we call a tile tree, operations (1) and (2) can be implemented in $O(n \log (\max\{n,k\}))$ time, where n is the cardinality of D. Operations (3) and (4) require at most $O(n \log (\max\{n,k\})+s)$ time, where s is the number of intervals listed. The tile tree occupies $O(n)$ space.

Using a previously-known plane-sweep technique, the operations above can be used to implement off-line algorithms to list all intersecting pairs of rectangles within a set of n aligned rectangles in $O(n)$ space and $O(n \log n + s)$ time.

1. Introduction

Many computer applications involve sets of aligned rectangles: rectangles with their sides parallel to the co-ordinate axes. In such applications it it frequently important to know which rectangles touch or overlap which other ones. For example, the simplest items of integrated circuit mask geometry are aligned rectangles. In order to do machine-aided geometric design rule checking or electrical circuit extraction, one needs algorithms that answer questions about which rectangles touch and overlap.

We present here a new data structure and associated algorithms for listing all intersecting pairs of rectangles within a set of n aligned rectangles in $O(n)$ space and $O(n \log n + s)$ time, where s is the length of the output list. This result derives from a more basic result about time-varying sets of linear intervals, which is presented first. In each case there are two dimensions involved. With the rectangles, both dimensions are spatial, and there is no temporal dimension. With the time-varying sets of linear intervals, a spatial dimension is exchanged for a temporal sequence.

Let D be a dynamic set of intervals over the set $1,2,...,k$ of integers. Consider the following interval operations applied to D:

InsertInterval[x,y]: Insert an interval [x,y] into D.

DeleteInterval[x,y]: Delete an interval [x,y] from D.

ListIntervalsContaining[u]: Given a test integer u, list those intervals [x,y] in D that contain u.

ListIntervalsOverlapping[u,v]: Given a test interval [u,v], list those intervals [x,y] in D for which [x,y] $\cap$ [u,v] is non-empty.

We present a new data structure, which we call a tile tree, for representing D, and a set of algorithms that operate on the tile tree to implement the interval operations above. We then argue that the tile tree requires space linear in n, the cardinality of D, and that the interval operations each require time at most $O(n \log \max\{n,k\} + s)$, where s is the length of any output list produced. Finally we complete the analogy between intervals and aligned rectangles.

2. Representing Intervals as Points

We define the one-dimensional closed interval [x,y], where $1 \leq x \leq y \leq k$, as the set of all integers between the integer x and the integer y, including x and y. Any such interval can be represented by a grid point (x,y) in a kxk square. Because $x \leq y$, this representative point must lie on or above and to the left of the diagonal line $x = y$. A set of intervals can thus be represented as a set of points in the upper left half of the square, as depicted in Figure 1.

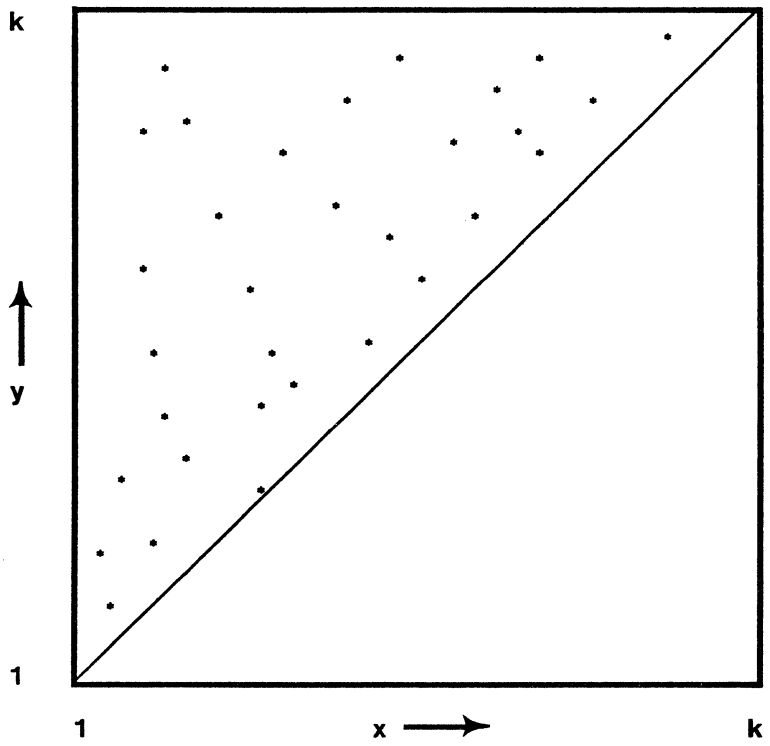

Figure 1

An interval [x,y] contains the integer u if and only if $x \leq u$ (so (x,y) lies to the left of (u,u)) and $u \leq y$ (so (x,y) lies above (u,u)). Therefore the intervals [x,y] containing any point u are exactly those whose representative points (x,y) lie both above and to the left of the point (u,u) in the square.

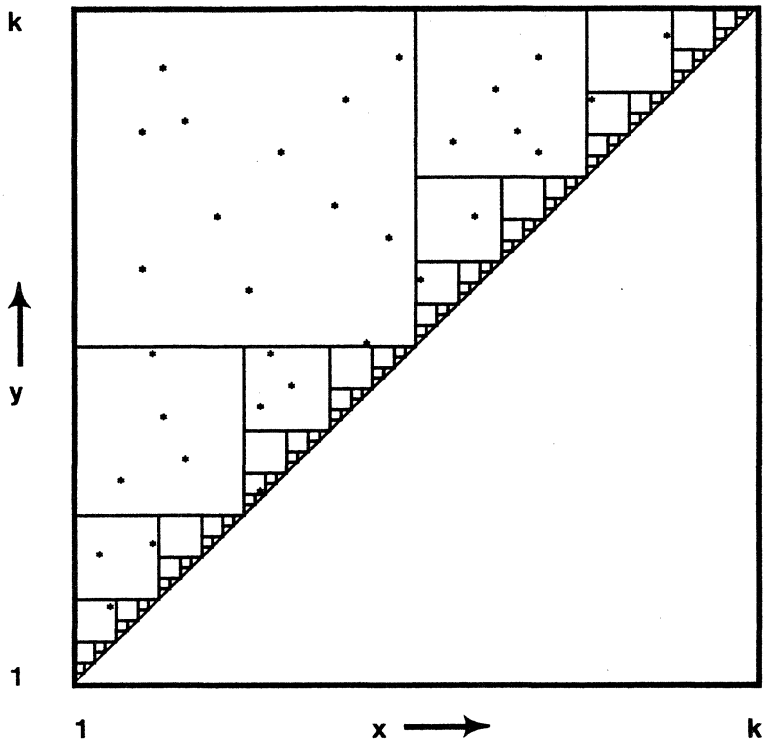

Figure 2

How can the **ListIntervalsContaining** operation enumerate its output intervals rapidly enough? This is the central question. Our answer is based on tiling the upper-left half of the square with a fixed set of ever-smaller square tiles, as shown in Figure 2. Figure 2 shows the ideal case where k is one less than a power of 2, but this restriction is not necessary. In Figure 2 the small triangles lying along the diagonal are assumed to have sides of length 1, and therefore to contain no points (x,y) where x and y are integers. Notice that each tile has its lower-right corner on the diagonal line $x=y$. Enumeration of the points above and to the left of any point (u,u) amounts to enumerating subsets of points in only those tiles that are cut either by the line $x=u$ or by the line $y=u$. Within such a cut tile, the points (x,y) to enumerate are either those for which $x \leq u$ or those for which $y \geq u$. A vital observation is that a single **ListIntervalsContaining** operation never requires that both these conditions be tested in the same tile. Figure 3 suggests how this enumeration might be carried out.

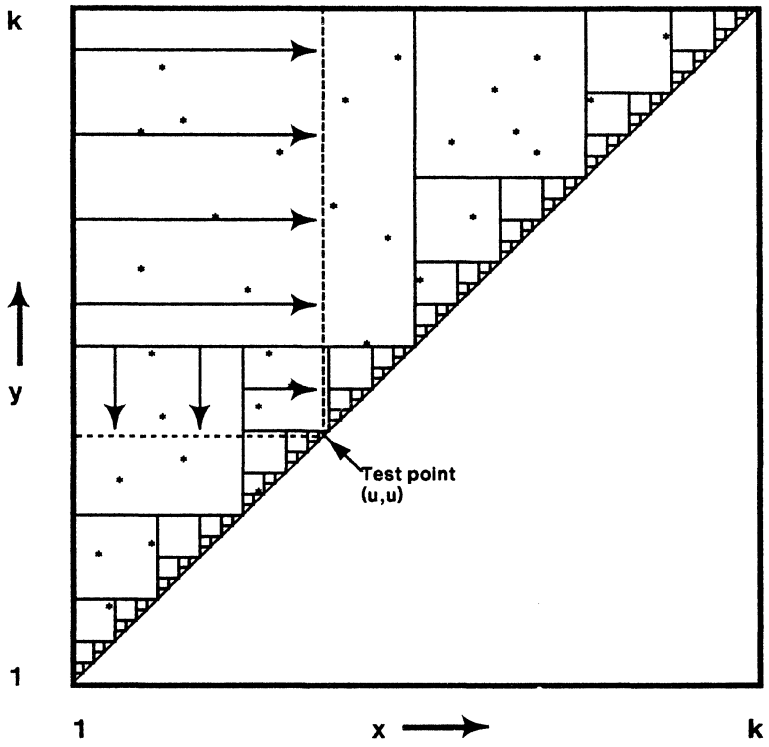

Figure 3

In the **ListIntervalsOverlapping** operation the intervals [x,y] intersecting the test interval **[u,v]** are exactly those whose representative points (x,y) lie above and to the left of the point (v,u) in the (lower-right half of the) square. These points divide into two disjoint subsets: those above and to the left of the point (u,u), and those to the right of u but to the left of v, as shown in Figure 4. The **ListIntervalsContaining** operation can enumerate the first subset. The second subset can be enumerated by a standard one-dimensional range query, for which data structures and algorithms with sufficient performance are well known [1].

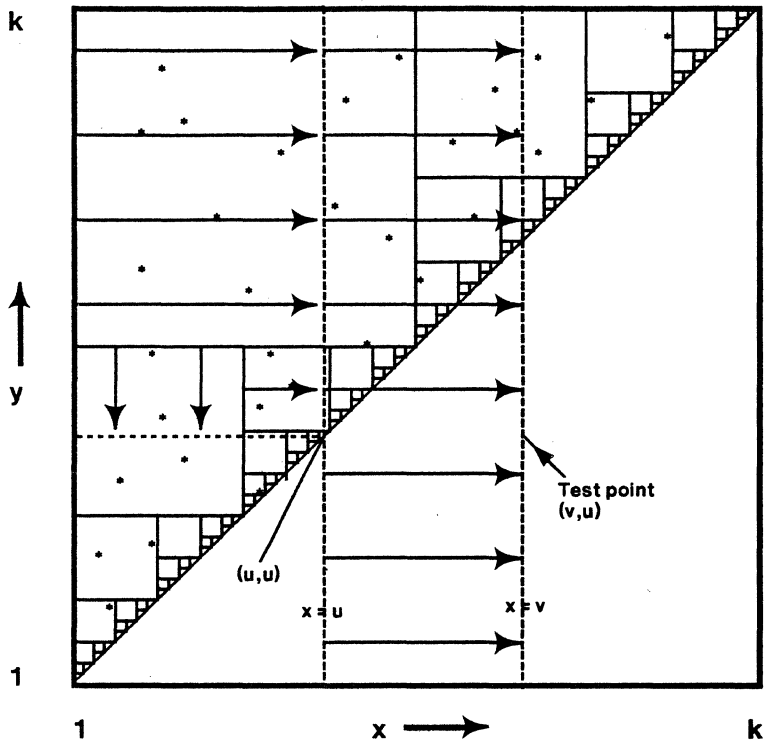

Figure 4

3. The Tile Tree

A *tile tree* is a particular storage representation for a dynamic set D of intervals, organized to permit rapid execution of the four basic interval operations.

3.1 Tile Keys

Every tile has exactly one point, its lower-right corner, on the diagonal line. Conversely every point on the diagonal line is the lower-right corner of some tile. We use this one-to-one mapping to assign a unique integer key, the x (or y) co-ordinate of its lower-right point, to each tile.

The key of a point (x,y) is defined as the key of the tile containing it. The key of an interval [x,y] is defined to be the key of the point (x,y). And finally, the key of a node in the tile tree is defined to be the key of the tile that the node represents.

These keys are easy to compute. The key of an interval [x,y] is that integer z in the interval whose binary representation contains the longest suffix of 0's. More formally, suppose $x = x_p2^p+...+x_12^1+x_0$ and $y = y_p2^p+...+y_12^1+y_0$, where $x_i,y_i\varepsilon\{0,1\}$, and $x\neq y$. If q^* is the smallest q such that x_q is different from y_q, then $key(x,y) = x_p2^p+...+x_{(q^*+1)}2^{(q^*+1)}+2^{q^*}$ (or equivalently $y_p2^p+...+y_{(q^*+1)}2^{(q^*+1)}+2^{q^*}$).

3.2 Tile Tree Structure

The main structure of a tile tree, shown in Figure 5, is a binary search tree of tile nodes that mimics the tile structure of Figure 2. A tile node t_2 is in a subtree of tile node t_1 whenever the corresponding tile t_2 is smaller in area than tile t_1, and there is no larger tile between them. This is equivalent to saying that the binary representation of the key of tile t_1 shares a common prefix with that of tile t_2, and after that prefix it is all 0's. In that event, tile node t_2 is in tile node t_1's left subtree if $key(t_2)<key(t_1)$, and in its right subtree otherwise.

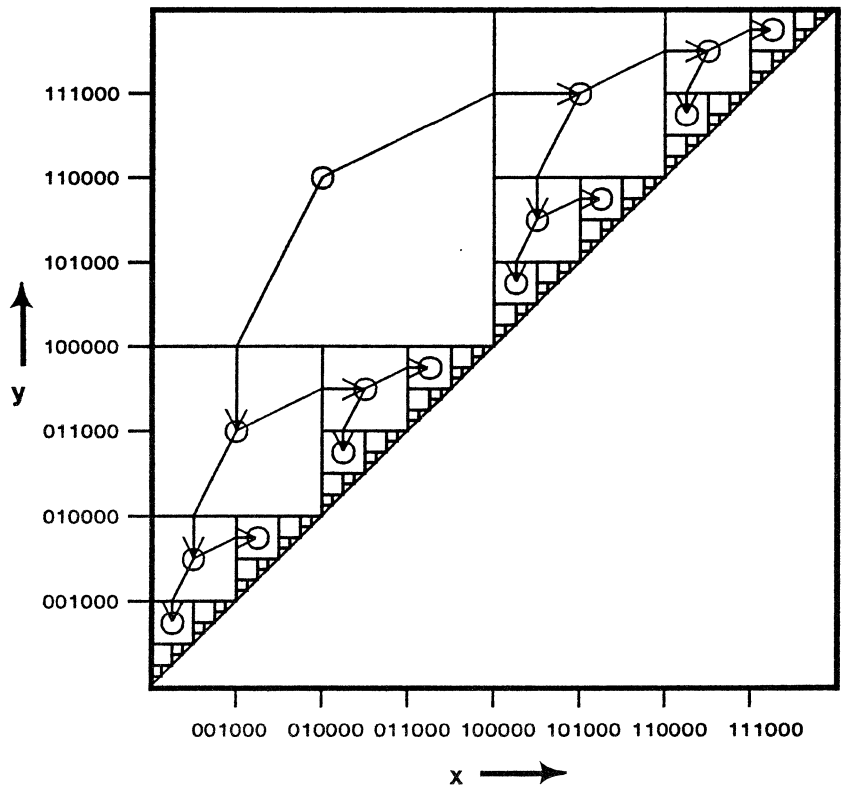

Figure 5

Tile nodes that contain no representative points of intervals in D need not be represented explicitly in the tile tree. In fact, it is easy to imagine situations of growth and shrinkage in D where if empty tile nodes were not pruned from the tile tree then the tile tree would require space more than linear in the size of D. Therefore the tile tree is further constrained to satisfy a *density condition*: every node in the tile tree, except the root node, whose tile contains no representative points of intervals in D must have non-empty both left and right subtrees. (This condition is suggested by the Patricia algorithm [1].) The density condition limits the total number of nodes in the tile tree below twice the number that contain representative points of intervals in D. A tile tree satisfying the density condition is shown in Figure 6.

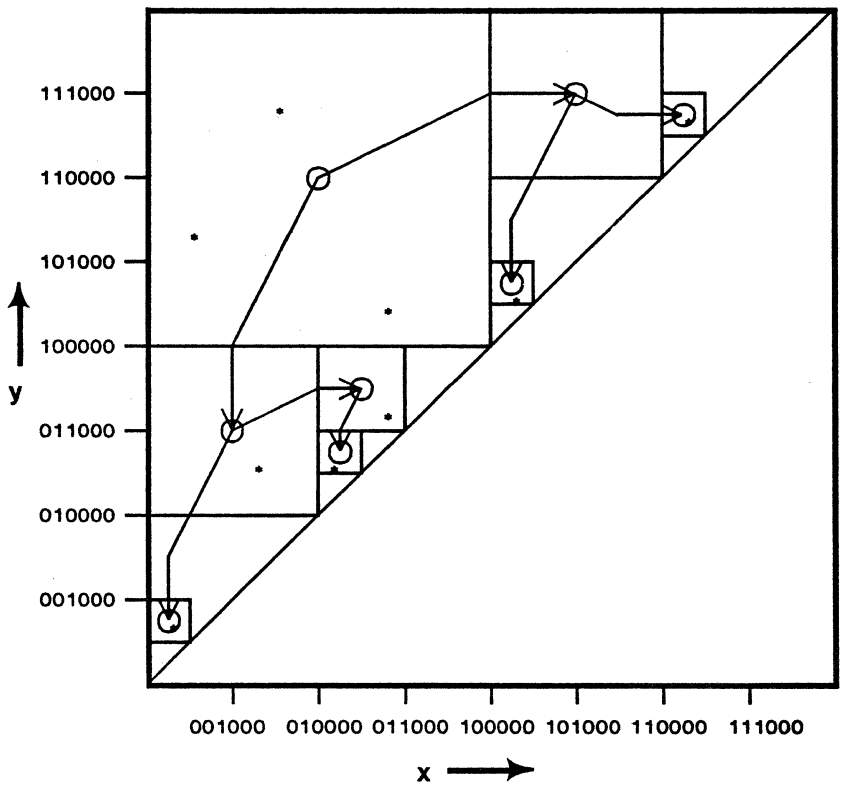

Figure 6

3.3 Tile Node Internal Structure

A tile node must contain (or refer to) information about every interval [x,y] in D whose representative point lies within its tile. Meeting the performance goals for **InsertInterval** and **DeleteInterval** requires that points be added to and removed from tile nodes quickly. However, the performance goal for **ListIntervalsContaining** requires enumerating the points in a tile node, sometimes increasing by x and other times decreasing by y, in real time (constant time per enumerated element).

There are several known structures that support random insertion and deletion in logarithmic time and yet permit ordered enumeration in real time. One is the very complicated finger tree [2], later simplified in [3]. Such complex structures overkill the present problem, which requires real-time enumeration only from the beginning, not from an arbitrarily-chosen place. More reasonable for this problem is a structure that we call an *endpoint list*, made up of a balanced binary search tree (for example, an AVL or 2-3 tree[1] or a red-black tree[7]) combined with an additional doubly-linked list threaded through the tree nodes in numeric order. The added linked list permits ordered enumeration of the nodes of the tree in real time, yet its maintenance adds at most constant time to tree updating operations.

Associated with every tile node are two endpoint lists, each containing an element for every interval in D whose representative point lies in the tile. One list is ordered by the x co-ordinates of its representative points, corresponding to the lower endpoints of intervals in D. The other list is ordered by the y co-ordinates of its representative points, corresponding to the upper endpoints of intervals in D. Actually the second list is ordered by -y, so that enumeration will happen in order of decreasing y.

3.4 Auxiliary List

ListIntervalsContaining requires the enumeration of those intervals [x,y] in D whose left endpoints x lie in a half-open test interval (u,v]. The performance goal for this operation necessitates real-time enumeration, but allows an additional time overhead logarithmic in the number of elements in D. This performance is achieved using an auxiliary list, implemented as an endpoint list as defined above, containing all the intervals in D keyed by their left endpoints.

This auxiliary list is not strictly necessary. All the information necessary for **ListIntervalsContaining** is contained in the tile tree, and **ListIntervalsContaining** can be programmed to run within its time bound. Without the auxiliary list, however, the algorithm is more difficult to program and to reason about.

4. Algorithms

Throughout these algorithms the parameter T will stand for the dense tile tree representing the dynamic set D of intervals.

CreateEmptySet[T]:

(0.1) Establish T as the tree consisting of the single tile node whose key is $2^{\text{floor}[\log_2 k]}$, and whose left and right subtrees and left and right endpoint lists are empty. Also establish the auxiliary list associated with T as empty.

InsertInterval[T,x,y]: To insert an interval [x,y],

(1.1) Search within T for the tile node t whose key is key(x,y), using standard (Patrician [1]) digital tree search. If t exists, skip to step (1.3). Otherwise let t^* be the least upper bound between t and any node t' present in T. If a node with key t* exists in T, skip to step (1.2). Otherwise splice a new tile node with key t* and empty endpoint lists into the proper place in T.

(1.2) Create a new tile node t whose key is key(x,y) and whose endpoint lists are empty as a son of node t*.

(1.3) Add a node for [x,y] to each of the two endpoint lists of t. The key of the node added to the left endpoint list will be x, while the key of the node added to the right endpoint list will be -y.

(1.4) Add a node for [x,y] with key x to the auxiliary list associated with T.

DeleteInterval[T,x,y]: To delete an interval [x,y],

(2.1) Find the tile node t whose key is key[x,y]. Let node t^* be the father of node t in T.

(2.2) Delete a node for [x,y] from each of the two endpoint lists of t. As with insertion, the key of the node deleted from the left endpoint list will be x, while the key of the node

deleted from the right endpoint list will be -y.

(2.3) If the endpoint lists of node t are non-empty, or if t has tile nodes in both its left and right subtrees, or if t is the root of the tile tree, then skip to step (2.5).

(2.4) Remove node t from T. If t^* has tile nodes in both its left and right subtrees, or if t^* is the root of the tile tree, then skip to step (2.5). Otherwise, splice node t^* out of T.

(2.5) Using the key x, delete a node for [x,y] from the auxiliary list associated with T.

ListIntervalsContaining[T,u]: Given a test point u, to list all intervals [x,y] in D that contain u,

(3.1) Search downward in the tile tree, using standard digital tree search, for the tile node t' with key u. At each tile node t along the way, test whether t' should be in the left or right subtree of t':

(3.1a) If t' should be in the left subtree of t, or if t=t', then list the nodes [x,y] in the left endpoint list of t in (increasing) x order as long as the keys x are no greater than u.

(3.1b) If t' should be in the right subtree of t, then list the [x,y] in the right endpoint list of t in (decreasing) y order as long as the keys y are no less than u.

ListIntervalsOverlapping[T,u,v]: Given a test interval [u,v], to list all intervals [x,y] in D that contain any point in the interval [u,v],

(4.1) Using the **ListIntervalsContaining** algorithm above, list all intervals [x,y] in the tile tree that contain the point u.

(4.2) Using the auxiliary list associated with T, list any intervals [x,y] whose left endpoint x lies in the half-open interval (u,v].

5. Worst-Case Analysis

5.1 Space

The performance of all of the algorithms depends on the depth of the tile tree, which in the worst case is the length of the binary representation of the tile node keys.

An tile tree containing n intervals with endpoints in the set $\{1,2,...,k\}$ requires $O(n \log (\max\{n,k\}))$ bits of storage. In standard terminology, this is $O(n)$ locations of storage, each capable of containing $O(\max\{n,k\})$ different values.

To demonstrate this we must count nodes. Each interval in D might have its very own unique tile corner, not shared with any other interval in D. If that happened, there would be at least as many tile nodes as intervals in D. Moreover, if all those tile nodes were leaves (instead of internal nodes), then at most that many (minus one) additional empty tile nodes might be required as internal nodes in the tile tree. This upper bound on the number of empty tile nodes exists because steps (1.2)-(1.3) of **InsertInterval** and steps (2.3)-(2.4) of **DeleteInterval** enforce the density condition. Each interval in D is represented in by an element in each of three endpoint lists: the lower endpoint list of a tile node t, the upper endpoint list of t, and the auxiliary list. Thus n intervals might require $2n$ tile nodes and $3n$ endpoint list elements. Each of these can be represented in $O(\log(\max\{n,k\}))$ bits.

5.2 Time

Using previous results about AVL trees [1], it is straightforward to show that **InsertInterval** and **DeleteInterval** can be done in $O(\log(\max\{n,k\}))$ primitive operations.

ListIntervalsContaining can be done in $O(\log(\max\{n,k\})+s)$ primitive operations, where s is the number of intervals listed. The search for the tile node with key u (step (3.1)) will encounter at most $\log k$ tile nodes, since $\log k$ is the maximum possible depth of the tile tree. While visiting each of these nodes, **ListIntervalsContaining** scans an endpoint list from its beginning to the point where a comparison condition fails. Each element in this scan, except possibly the last one, represents an additional interval [x,y] containing the point u, and thus contributes to s. Therefore the amount of time at a tile node not chargeable to listing output intervals is bounded by a constant.

ListIntervalsOverlapping can also be done in $O(\log(\max\{n,k\})+s)$ primitive operations, because it uses **ListIntervalsContaining** as a subroutine, and then uses additional time $O(\log n + s)$ to make a random access into the auxiliary list and then from that point lists output intervals until a simple comparison condition fails.

6. Aligned Rectangles

An algorithm to enumerate all intersecting pairs of aligned rectangles combines tile trees with the plane sweep technique first proposed by Shamos and Hoey [4]. This plane sweep techinque simulates the motion of a line of constant x across the plane, and considers the sequence of cross-sectional slices induced by this line.

For aligned rectangles these cross-sectional slices are sets of intervals in y. As the sweep line moves onto a new rectangle, that rectangle's y interval is added to the set D. As the sweep line moves off a rectangle, its y interval is removed from D.

The intersections in which a rectangle participates can be enumerated at either edge. Of course, not all of the intersections in which a given rectangle participates are apparent at its edges. For example, if R_1 completely contains R_2, then D will contain no trace of R_2 when the sweep line encounters the edges of R_1. However, the intersection is discovered as the sweep line passes the edge of $R_1 \cap R_2$, which always corresponds either to an edge of R_1 or to an edge of R_2. Thus for this simple application there is no need to separate insertion/deletion from listing, because they can always be done together.

A more complex application like circuit extraction from IC masks might involve rectangles of several colors, and be concerned only with intersections between rectangles of different colors. For this purpose one could have a different tile tree for each color. As the sweep line passes the left edge of a red rectangle, for example, the corresponding red interval would be inserted into the red tile tree, while the same interval would be used in a **ListIntervalsOverlapping** operation on every non-red tile tree. In this application the separation of insertion/deletion from listing is important. Notice further that arbitrary intersection patterns of red with red rectangles do not increase the time beyond $O(n \log n)$. The s term counts only the number of *inter-color* intersections.

We now analyze the performance of the rectangle intersection algorithm somewhat more carefully. The sweep technique requires an initial sort on x. For n rectangles this takes $O(n)$ space and $O(n \log n)$ time. A second sort can be used if necessary to normalize the y values to $\{1,2,...,2n\}$. Then the tile tree operations can be done in $O(n)$ space and $O(\log n + s)$ time apiece, because k is linear in n. Each rectangle causes one **InsertInterval**, one **DeleteInterval**, and one **ListIntervalsOverlapping** operation. Moreover each rectangle intersection is discovered by exactly one ListIntervalsOverlapping operation, and therefore contributes to the s of that operation. Thus the overall time performance is $O(n \log n + s)$. This performance is within a constant factor of the best possible worst-case performance. These space and time bounds have been achieved separately before [5], but not together.

7. Observations

★ There is nothing sacred about the tiling presented here. It is simply a fixed balanced structure in the tile tree. The tile tree structure reflects the binary digits of the interval endpoints. Guibas and Saxe have discovered a comparason-based analog of the tile tree, whose performance differs from that of the tile tree by at most a constant factor. In so doing they have solved the additional problem of how to alter the tiling structure in response to changing populations of intervals. This alteration requires additional machinery, space, and time that make the comparason-based tree less attractive than the digital one for many practical applications.

★ What about other tests, like interval containment? Can the set of all intervals totally containing a test interval be listed in time $O(\log (\max\{n,k\})+s)$, where s is the number of such containments found? (If this test were possible even restricted to a single tile, it would be possible in general.)

★ The present algorithm is an on-line one dimensional algorithm. Its conversion to a two dimensioned algorithm sacrifices its on-line property, in effect trading a temporal dimension for a spatial one. Perhaps there exists a multi-dimensional analog of the tile tree. Such a structure would be very useful in interactive graphical design of integrated circuits and printed circuit boards.

Acknowledgments

I thank Jon Bentley for pointing out the gaping hole in my first attempt to solve this problem, Jurg Nievergelt for the didactic mapping between intervals and the unit square, and Jim Saxe for the clean separation between **ListIntevalsContaining** and **ListIntervalsOverlapping.** I also thank John Warnock, Leo Guibas, Bob Sedgewick, Mark Brown, and others for discussions that led to the present algorithms and presentation.

References

1. Knuth, D.E., *The art of computer programming*, vol. 3 ("Sorting and Searching"), sections 6.2 and 6.3, (Addison-Wesley) 1973.

2. Guibas, L.J., McCreight, E.M., Plass, M.F., and Roberts, J.R., A new representation for linear lists, *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pp. 49-60. (ACM), 1977.

3. Brown, M.R., and Tarjan, R.E., Design and analysis of a data structure for representing sorted lists, *SIAM J. Comp.* (to appear).

4. Shamos, M.I., and Hoey, D., Geometric intersection problems, *17th Annual Symposium on Foundations of Computer Science*, pp. 208-215. (IEEE), 1975.

5. Bentley, J.L., and Wood, D., An optimal worst-case algorithm for reporting intersections of rectangles, Technical Report, Dep't of Computer Science, Carnegie-Mellon University, July 1979.

6. Nievergelt, J., and Preparata, F.P., Plane-sweep algorithms for intersecting geometric figures, Technical Report in preparation, Institut fuer Informatik, ETH, Zurich.

7. Guibas, L.J., and Sedgewick, R., A dichromatic framework for balanced trees, *19th Annual Symposium on Foundations of Computer Science*, pp. 8-21. (IEEE), 1978.

XEROX

Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles

by Edward M. McCreight

CSL-80-9

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

XEROX® is a trademark of XEROX CORPORATION

Printed in U.S.A.